Mark A. O'Neill

Faster Than Fast Fourier

The fast Hartley transform is twice as fast as the fast Fourier and uses only half the computer resources

Joseph Fourier left to mathematics a rich legacy: The transform that bears his name processes many of today's audio and electromagnetic signals. Image processing and digital filtering of signals use the Fourier transform or its faster descendant.

However, the fast Fourier transform (FFT) requires a large amount of computer resources. The fast Hartley transform (FHT) can accomplish the same results faster, using fewer resources. First described by Ronald Bracewell (see reference 1), the FHT handles many of the jobs now done by the Fourier transform.

Both the FHT and the FFT let you map a continuous signal over time onto a frequency function. The Fourier transform maps a real function of time $X(t)$ to a complex function of frequency, $F(f)$.

The Hartley transform maps a real function of time $X(t)$ onto a real function of frequency $H(f)$. Since the Hartley frequency function is real, you need only single arithmetic operations to compute it.

Compare this to the many arithmetic operations required of the complex Fourier frequency function (four operations for a complex multiply or divide, and two for complex addition or subtraction).

Furthermore, real data arrays require only half the memory storage of complex data arrays. This means that the Hartley transform will require considerably less memory for a given data set than the Fourier transform. Therefore, the Hartley transform will be distinctly faster and use less memory than the conventional Fourier transform in digital filtering and image enhancement applications where you have to process large amounts of data.

A Definition of the Hartley Transform

Equation (1) shows the analytical form of the Hartley transform, and (2) shows its inverse transform, used to map the frequency function back into the time domain:

$$H(f) = \frac{1}{2\pi} \int_{-\infty}^{\infty} x(t) \operatorname{cas}(2\pi ft)\, dt, \qquad (1)$$

$$X(t) = \int_{-\infty}^{\infty} H(f) \operatorname{cas}(2\pi ft)\, df, \qquad (2)$$

where $\operatorname{cas}(2\pi ft) = \cos(2\pi ft) + \sin(2\pi ft)$.

The function $\operatorname{cas}(2\pi ft)$ was introduced by R.V.L Hartley (see reference 2), who first proposed the Hartley transform in 1942.

You can see that these equations are very similar to those of the Fourier transform and its inverse:

$$F(f) = \frac{1}{2\pi} \int_{-\infty}^{\infty} X(t)\, e^{-j2\pi ft}\, dt, \qquad (3)$$

$$X(t) = \int_{-\infty}^{\infty} F(f)\, e^{j2\pi ft}\, df, \qquad (4)$$

where $e^{j2\pi ft} = \cos(2\pi ft) + j\sin(2\pi ft)$, and $e^{-j2\pi ft} = \cos(2\pi ft) - j\sin(2\pi ft)$. (These are known as Euler's formulas.)

Note that I'm using the electrical engineering convention of labeling the imaginary unit i as j. The principal difference between the two functions is that the real function $\operatorname{cas}(2\pi ft)$ in the Hartley transform replaces the complex exponential term $e^{\pm j2\pi ft}$ in the Fourier transform pair.

The Hartley and Fourier transform functions in (1) through (4) deal only with continuous variables. As is often the case with computer data in the real world, signals are sampled at discrete intervals of time or for a specific interval. Fortunately, you can define a discrete transform that can represent a quantized continuous signal, or a signal of limited duration. The discrete forms of the Hartley transform pairs are

$$H(f) = \frac{1}{N} \sum_{t=0}^{N-1} F(t) \operatorname{cas}(2\pi ft/N) \qquad (5)$$

and

$$X(t) = \sum_{f=0}^{N-1} H(f) \operatorname{cas}(2\pi ft/N). \qquad (6)$$

Again, note the similarities to the Fourier transform pairs:

$$F(f) = \frac{1}{N} \sum_{t=0}^{N-1} X(t)\, e^{(-j2\pi ft/N)}, \qquad (7)$$

$$X(t) = \sum_{f=0}^{N-1} F(f)\, e^{(j2\pi ft/N)}. \qquad (8)$$

The Fast Algorithm

As it stands in (5) and (6), computation of the discrete Hartley transform presents an analogous problem to the computation of the discrete Fourier transform. That is, you have to perform N^2 arithmetic operations to compute the discrete Hartley transform of an N-element data set.

A classic paper by Cooley and Tukey (see reference 3) in 1965 led to the development of a fast algorithm for the machine computation of a complex Fourier series.

Essentially, the FFT uses a permutation process to bisect the data until data pairs are reached. Calculating the Fourier trans-

continued

form of such data pairs is trivial (i.e., rapid).

The idea behind the permutation process is that it's faster to split the data into pairs, compute the transform of the pairs, and recombine these to make the entire transform rather than to compute the transform for the complete data set.

Permutation is particularly fast when the amount of data is large. If you superimpose all such two-element pairs using a process sometimes referred to as the "butterfly" (due to the appearance of the diagram of the data flow; see figure 1), you can compute the Fourier transform of the input data set. It takes approximately $N \log(N)$ seconds to compute the transform of an N-point data set.

Bracewell (see reference 4) has shown that you can employ a similar methodology in the case of the Hartley transform. Again, you use the permutation process to bisect the data until you get data pairs.

The Hartley transform of a data pair (a, b) is $\frac{1}{2}(a + b, a - b)$, and the computation of such pairs is trivial. You can also superimpose these two-element sequences to calculate the Hartley transform of the input data set. However, to do so requires a formula that expresses a complete discrete Hartley transform (DHT) in terms of its half-length subsequences.

Bracewell has shown by application of the Shift and the Similarity theorems that (9) expresses the general decomposition formula for the DHT. This general decomposition formula generates the desired DHT by bisecting the data.

Put another way, it's the rule used to generate the elements to be used in the butterfly computation of the transform. You can apply similar methodology to the Fourier transform to yield the decomposition formula given in (10):

$$H(f) = H_1(f) + H_2(f)\cos(2\pi f/N_s) + H_2(N_s-f)\sin(2\pi f/N_s), \quad (9)$$

$$F(f) = F_1(f) + F_2(f)e^{j2\pi f/N_s}, \quad (10)$$

where N_s is the number of elements in the half-length sequence, and thus $N_s=N/2$ for a data set of N elements. Figure 1 shows a complete butterfly diagram for computing the FHT for an eight-element data set.

The decomposition formula for the FHT differs from the FFT in one important respect: The elements multiplied by the trigonometric terms are not symmetric. In the FFT decomposition formula of (10), terms multiplied by the trigonometric coefficients involve terms only in $F(f)$. In the FHT decomposition formula of (9), both $H(f)$ and $H(N_s-f)$ have sine coefficients. This asymmetry becomes apparent when you express the discrete transforms as matrix operations: FFT matrix terms are symmetric about the matrices' leading diagonal, while corresponding terms for the FHT are asymmetric.

This introduces some computation problems, because asymmetric matrix processing is difficult to implement. You can deal

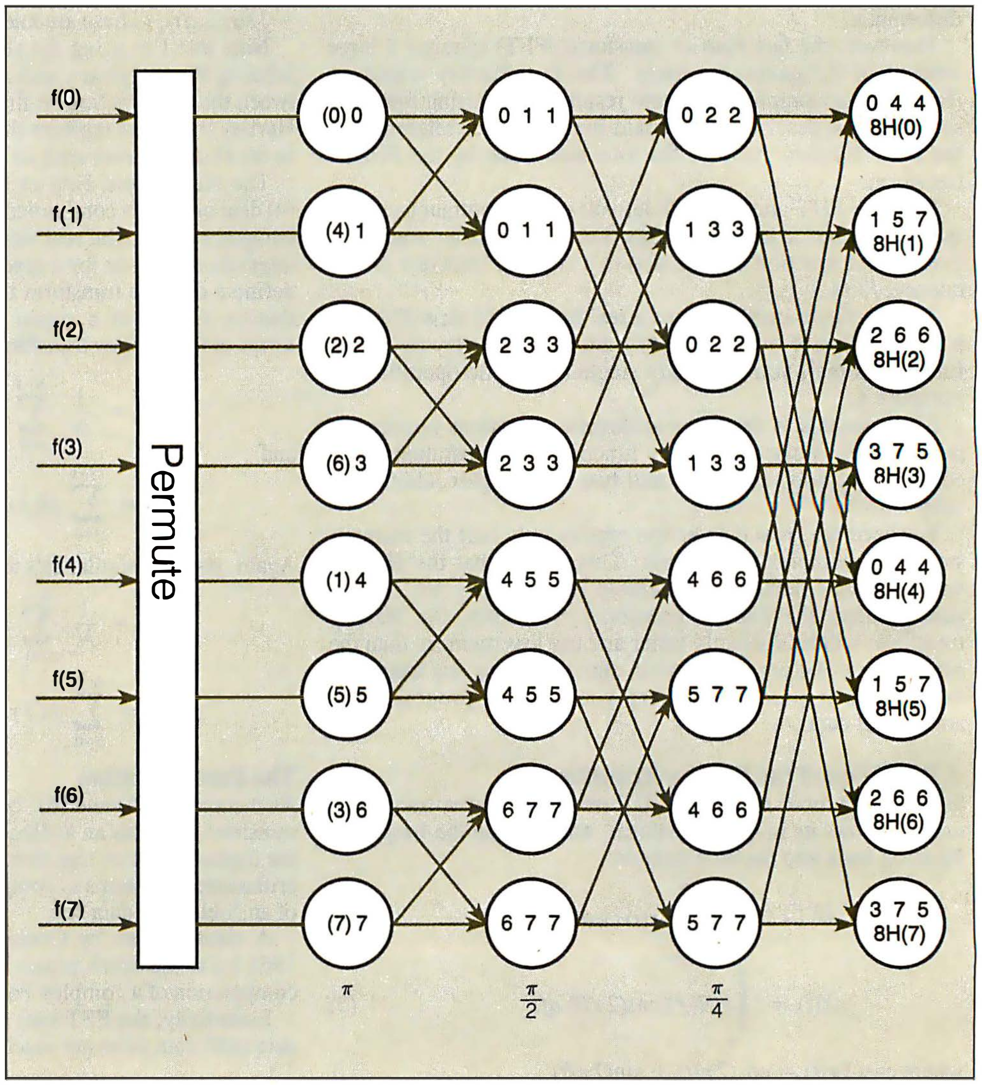

Figure 1: *The butterfly diagram for the fast Hartley transform for an eight-element data set. In the first column of cells, the indexes show the effects of permutation on the data elements. The next three columns represent the data elements as superimposed by the decomposition formula for the butterfly computation. These cells show the three indexes of the elements. You can see the effects of retrograde indexing for an eight-element data set in indexes for the third and fourth iteration of the butterfly. Radians at the bottom of the diagram give the angular spacing between trigonometric coefficients.*

with this asymmetry by using an independent variable as an index for the elements multiplied by the sine coefficients. This index decreases while the other indexes increase; this behavior is called retrograde indexing. Bracewell gives descriptions of the FHT and FFT algorithms using matrix formalism (see reference 5).

You can obtain the inverse Hartley transform by applying the FHT algorithm again to its own output, thus regenerating the input data. This means that you can use the same program code to compute the transform and its inverse. However, there is a slight asymmetry between the FHT and its inverse. In the case of the time-to-frequency transform, you need to scale the result of the butterfly computation. That is, for an input data set of N elements, you must divide the output of the butterfly by N to obtain the discrete Hartley transform.

You don't need to do this for frequency-to-time transform; consequently, the butterfly computation itself constitutes the inverse transform. It is not difficult to add a small amount of code to control whether scaling should be applied during a conversion.

Comparing the FFT and FHT Algorithms

You can use the FHT algorithm for many of the applications for which you would now use the FFT algorithm. These include convolution and deconvolution (used to remove artifacts introduced into data by imperfections in the sensors), and the generation of power spectra for filtering. You can also obtain the Fourier transform itself from the Hartley transform. In fact, it is often faster to generate the Fourier transform and power spectrum with the FHT than with the FFT, because computing the butterfly using real rather than complex quantities requires fewer floating-point operations. You assemble the real and imaginary parts of the FFT at the end of the calculation using the equations

$$F_r = H(f) + H(N-f)$$

and

$$F_{im} = H(f) - H(N-f),$$

where F_r is the real portion of the complex Fourier transform, F_{im} is the imaginary portion, and N is the number of elements in the data set.

You can calculate the power spectrum directly from the Hartley transform using the equation

$$P_s(f) = [H(f)^2 + H(N-f)^2]/2,$$

where P_s is the power spectrum.

The theorem for convolving a pair of functions is almost identical, whether you are considering the Hartley transform or the Fourier transform. Again, the FHT should prove superior to the FFT in terms of speed for any given implementation. Equation (11) summarizes the theorem for the convolution for the Hartley transform, (12) for the Fourier transforms.

$$f_1(t) \circledast f_2(t) = H_1(f)H_{2e}(f) + H_1(-f)H_{2o}(f) \qquad (11)$$

and

$$f_1(t) \circledast f_2(t) = F_1(f)F_2(f). \qquad (12)$$

The $\circledast$ symbol denotes the convolution operation.

The subscripts e and o in (11) denote the even and odd parts of the Hartley transform. Note that if one of the functions being convoluted is either even *or* odd, then the form of the convolution theorem for the Hartley transform reduces to the particularly simple form indicated in (13):

$$f_1(t) \circledast f_2(t) = H_1(f)H_2(f). \qquad (13)$$

In practical applications, many of the convolution functions are even. For example, the Gaussian function, used in image-enhancement work, is an even function. We can take advantage of the nature of these functions to use the computational shortcut provided by (13).

Spectral Leakage

As with the FFT algorithm, the FHT algorithm will produce an erroneous frequency function if the data set to be transformed does not smoothly approach 0 at both ends of its range. Such spectral leakage is undesirable in many cases. You can reduce it

continued

Listing 1: *The code fragment from* `hartley.pas` *that shows the complete Hartley transform as implemented in TML Pascal.*

```
{------------------------------------
   Fast Hartley transform routine ...
   version 1.00, Dated 22nd November 1986,

transform:= forward; Forward transform from
time frequency domain. transform := reverse;
Reverse transform from frequency to time
domain.

power_index: Index to which 2 must be raised
to generate a transform containing 'syze'
elements:
      if syze = 8  then power_index = 3;
      if syze = 16 then power_index = 4; etc
...

syze:  Number of element in the input data
array.
------------------------------------}

const
   datasize = 512;
type
   direction_type = (forward,reverse);
   data_array_type = array[1..datasize] of
real;
var
   dir, test_option: char;
   i,j,syze,iter,demo: integer;
   data_array: data_array_type;
   transform_direction: direction_type;

procedure fht(var data_array: data_array_type;
                    power_index,
                    syze: integer;
                    transform:direction_type);

var
   i,
   j,
   k,
   trg_ind,
   trg_inc,
   power,
   t_a,
   f_a,
   i_temp,
   section,
   s_start,
   s_end: integer;
   sne,csn: array[1..datasize] of real;
   accu: array[1..2,1..datasize] of real;
```

continued

```
{-------------------------------------
Permutation routine. This routine
reorders the data before the butterly
transform routine is called ...
-------------------------------------}

function permute(index: integer): integer;
var
  i,j,s: integer;
begin
   j := 0;
   index := index - 1;
   for i := 1 to power_index do
   begin
      s := index div 2;
      j := j + j + index - s - s;
      index := s;
   end;
   permute := j + 1;
end;

{-------------------------------------
Calculate the trigonometric functions
required by the FHT and store values.
For a N point transform, the trigno-
metric functions will be calculated
at intervals of Nths of a turn ...
-------------------------------------}

procedure trig_table(npts: integer);
const
   pi = 3.14159265;
var
   i: integer;
   angle,omega: real;
begin
   angle := 0;
   omega := 2 * pi / npts;
   for i := 1 to npts do
   begin
      sne[i] := sin(angle);
      csn[i] := cos(angle);
      angle := angle + omega;
   end;
end;

{-------------------------------------
Calculate the address of the retrograde
index for the sine term for the dual
place algorithm, if it is required ...
-------------------------------------}

function modify(power,s_start,s_end,index:
integer): integer;
begin
   if (s_start = index) or (power < 3) then
      modify := index
   else
      modify := s_start + s_end - index + 1;
end;
{-------------------------------------
   Butterfly transform an index pair ...
-------------------------------------}

procedure butterfly(trig_ind,i_1,i_2,i_3:
integer);
begin
   accu[t_a,i_1] := accu[f_a,i_1] +
           accu[f_a,i_2] * csn[trig_ind] +
           accu[f_a,i_3] * sne[trig_ind];
```

continued

by multiplying the data set to be transformed by a suitable window function before computing the transform.

All these window functions cause the data to smoothly approach 0 at the limits of its range. Stigall, Ziemer, and Hudec have reviewed the effects of these window functions on the power spectrum (see reference 6).

If you're interested in some experimentation, I've provided a number of these window functions below. The triangular window is a low-quality window function whose primary advantage is its speed. An example of its use is to filter noise from telemetry signals in real time.

Triangular window:
$W(n) = 2(n+0.5)/n$,
$W(N-n-1) = W(n)$,

where $n = 0,1, \ldots i, \ldots, N/2$.

The Hanning and Hamming windows feature better reduction in spectral leakage, but at the expense of speed. I've used these window functions to preprocess satellite image data before using the Hartley transform to correct the images for defects in the camera optics.

Hanning window: $W(n) = 0.5(1-\cos(2\pi(n+0.5)/N))$,

where $n = 0,1, \ldots i, \ldots, N-1$.

Hamming window: $W(n) = 0.54 - 0.46(\cos(2\pi(n+0.5)/N))$,

where $n = 0,1, \ldots i, \ldots, N-1$.

The Blackman window is for data sets where the Hamming/Hanning windows would not be suitable.

Blackman window: $W(n) = 0.42 - 0.5\cos(2\pi(n+0.5)/(N-1))$,

where $n = 0,1, \ldots i, \ldots, N-1$.

You can find additional information about spectral window functions in the papers by Harris (see reference 7) and Nutall (see reference 8).

The FHT program

The demonstrator program `hartley.pas` described below computes the Hartley transform and inverse Hartley transform of an N-element data set. The program was originally written in Acornsoft ISO Pascal for the BBC Microcomputer, but few machine-dependent Pascal extensions are used.

The routine has also been successfully ported to a Prime 9975 minicomputer, running Sheffield Pascal, and a Sun-3/160 workstation running International Standard Organization level 0 Pascal.

As an example of just how easy it is to port the fast Hartley routine to other machines, I have included the code for `hartley.pas` as written in TML Pascal for the Macintosh.

The `hartley.pas` program simply generates a suitable function to be transformed and then calls the `fht` procedure twice. Thus, the program calculates both the Hartley transform and its inverse transform. The procedure `fht` is the major routine in the program. Listing 1 is a code fragment of `hartley.pas` that contains the `fht` procedure.

The code within procedure `fht` complies with the ISO level 1 standard for Pascal, except that the program uses the underscore character in variable names. Consequently, you can lift the pro-

continued

```
    trig_ind := trig_ind + syze div 2;
    accu[t_a,i_2] := accu[f_a,i_1] +
             accu[f_a,i_2] * csn[trig_ind] +
             accu[f_a,i_3] * sne[trig_ind];
end;




{--------------------------------------
    Main program for the fast Hartley
    transform.
--------------------------------------}

begin

    power := 1;
    f_a := 1;
    t_a := 2;
    trig_table(syze);
    for i := 1 to syze do accu[f_a,permute(i)]
 :=
            data_array[i];

{--------------------------------------
    Start of the Hartley butterfly
    transform ...
--------------------------------------}

    for i := 1 to power_index do
    begin
       j := 1;
       section := 1;
       trg_inc := syze div (power + power);
       repeat
          trg_ind := 1;
          s_start := section * power + 1;
          s_end   := (section + 1) * power;
          for k := 1 to power do
          begin
             butterfly(trg_ind,j,j + power,
                    modify(power,s_start,s_end,j
                                  + power));
             trg_ind := trg_ind + trg_inc;
             j := j + 1;
          end;
          j := j + power;
          section := section + 2;
       until j > syze;
       power := power + power;
       i_temp := t_a;
       t_a    := f_a;
       f_a    := i_temp;
end;

{--------------------------------------
End of Hartley butterfly. The results are
scaled if necessary, and then placed in
back into the array data ...
--------------------------------------}

    case transform_direction of
            forward: for i := 1 to syze do
                                 data_array[i] :=
accu[f_a,i] / syze;
          reverse: for i := 1 to syze do
                                 data_array[i] :=
accu[f_a,i];
    end;
end;
```

cedure from the demonstration program and use it freely in your own applications.

However, a brief description of fht and its embedded subprocedures may be useful if you'd like to experiment with the program.

The procedure trig_table precalculates the trigonometric functions required by fht. For an input data set containing N elements, adjacent entries in this table are $2\pi/N$ radians apart. This precalculation of the trigonometric terms avoids redundant computations of these values if they are needed frequently during processing.

The procedure permute bisects the input data set progressively until data pairs are reached. The algorithm given is adapted from one described by Bracewell, who has also described faster permutation routines (see reference 5) that you can substitute if you want.

The procedure butterfly calculates the butterfly of a single index pair. This procedure requires three indexes to use as entry parameters. This takes into account the possibility of retrograde indexing for the data element multiplied by the sine factor. Where retrograde indexing is necessary, the procedure modify calculates the retrograde indexes used by the procedure butterfly.

The main program of the fht procedure first precalculates the trigonometric function tables and permutes the input data. The program then enters an iterative loop that computes the butterfly for the whole of the N-element input data set. You then scale the resulting output by a factor of N if you are computing the time-to-frequency transform.

When you have to deal with the complex form of the frequency transform, you can obtain Fourier transform from the FHT algorithm by operating on the output data of the procedure fht using the Pascal procedure get_FFT.pas, shown in listing 2. By the time you exit from this procedure, the real and imaginary parts of the Fourier transform will be in the arrays. They are passed via the dummy conformant array variables r_pt and i_pt, respectively.

A similar procedure in get_pwr.pas called get_power_spectrum (shown in listing 3) allows you to obtain the power spectrum from the Hartley transform. At exit from this procedure, the power spectrum will be in the array passed to the procedure via the dummy conformant array variable p_sp. You can use both of the above procedures in applications programs without any modification. These procedures conform to the ISO level 1 standard of the Pascal language, and should be usable directly on any computer that supports an ISO level 1 Pascal compiler.

The modifications required to compile these procedures with an ISO level 0 compiler are trivial. Basically, it is just a case of replacing all the conformant array parameters' procedure headers of get_fft and get_power_spectrum, with array parameters of fixed size.

Performance

The BBC Microcomputer implementation of the FHT program has proved to be fast and is currently running on a 6502A-based BBC Microcomputer with a second add-on 3-MHz 6502 processor, which executes the program. This hardware can compute a 256-point transform in about 32 seconds. This compares favorably with the 2 minutes it took Bracewell's original implementation of the FHT algorithm to accomplish a 256-point transform on an HP-85 system. You can expect even better performance from 16-bit microcomputers.

Table 1 shows the timings for running the FHT program on the Macintosh Plus and Mac II using TML Pascal and the Sun-3/160 workstation using ISO Pascal.

continued

Table 1: *Timings for the* `hartley.pas` *demonstrator program on several machines. For the Macintosh computers, TML Pascal version 2.50 was used, and times are to the nearest second. No 68881 coprocessor code was generated for the test, so the Mac II's 68881 was not used. You can expect better times if you use the 68881. For comparison, the results of the same code compiled on a Sun-3/160 workstation using ISO Pascal are shown. Times are in seconds.*

Number of points	Mac Plus FPU not used	Mac II FPU not used	Sun-3/160 FPU not used	Sun-3/160 68881 FPU used
2	0.00	0.00	0.00	0.00
4	0.00	0.00	0.02	0.00
8	0.00	0.00	0.04	0.00
16	1.00	0.00	0.08	0.02
32	1.00	1.00	0.18	0.04
64	2.00	1.00	0.34	0.06
128	6.00	2.00	0.70	0.08
256	12.00	3.00	1.48	0.18
512	27.00	6.00	3.60	0.40

Listing 2: *Pascal code to compute the Fourier transform from the FHT algorithm.*

```
Procedure get_FFT(var data:
array[l0..h0:s_int] of real;
 var r_pt: array[l1..h1:s_int] of real;
 var i_pt: array[l2..h2:s_int] of real;
 size: s_int);
 var i: s_int;
 begin
 i = 1;
while i<size do
 begin
 r_pt[i] := data[i] + data[size - i + 1];
i_pt[i] := data[i] - data[size - i + 1];
 i := i + 2;
 end;
end;
```

Listing 3: *Pascal code to compute the power spectrum from the FHT algorithm.*

```
 procedure get_power_spectrum
 (var data: array[l0..h0:s_int] of real;
 var p_sp: array[l1..h1:s_int] of real;
size: s_int);
 var i: s_int;
begin
 i := 1;
while i<size do
 begin
 p_sp[i] := (data[i] * data[i] +
      data[size - i + 1] *
      data[size - i]) / 2;
 i := i + 1;
 end;
 end;
```

Note that you can attain very high speeds with the Sun workstation, with its combination of a 20-MHz 68020 processor, 68881 floating-point coprocessor, and optimizing Pascal compiler.

New Limits to Explore

I hope this article has given you enough information to seriously consider using the Hartley transform for your signal processing needs. Since the transform uses only real functions, you don't need computationally expensive complex math to digitally filter or enhance a signal.

The elimination of complex numbers also reduces the amount of memory required to process a signal. Finally, since the Hartley transform uses fewer operations to process a signal, you will have fewer roundoff errors.

The FFT made a lot of what we call image processing possible by manipulating large amounts of data in a reasonable amount of time. The FHT offers better performance using less computational resources. With the same code you use to compute the transform, you can compute its inverse when necessary. It will be interesting to see what new uses will result from the expanded limits of processing that the Hartley transform provides. ■

Editor's note: *Source code listings that accompany this article,* `hartley.pas`, `get_FFT.pas`, *and* `getpwr.pas`, *are available on BIX, on BYTEnet, on disk, and in the Quarterly Listings Supplement. See page 3.*

REFERENCES
1. Bracewell, Ronald N. "The Fast Hartley Transform." *Proceedings of the IEEE*, vol. 72, no.8, p.1010, 1984.
2. Hartley, R. V. L. "A More Symmetrical Fourier Analysis Applied to Transmission Problems." *Proceedings of the IRE*, vol. 30, p. 144, 1942.
3. Cooley, J. W., and J. W. Tukey. "An Algorithm for the Machine Computation of Complex Fourier Series." *Mathematical Computing*, vol. 19, p. 297, 1965.
4. Bracewell, Ronald N. "The Discrete Hartley Transform." *Journal of the Optical Society of America*, vol. 73, p. 1832, 1983.
5. Bracewell, Ronald N. *The Fast Hartley Transform*. New York: Oxford University Press, 1986.
6. Stigall, P., R. E. Ziemer, and L. Hudec. "A Performance Study of 16-bit Microcomputer-Implemented FFT Algorithms." *IEEE Micro*, p. 61, November 1982.
7. Harris, F. J. "On The Use of Windows for Harmonic Analysis with Discrete Fourier Transforms." *Proceedings of the IEEE*, vol. 66, no. 1, p. 51, 1978.
8. Nutall, H. H. "Some Windows with Very Good Sidelobe Behaviour." IEEE transcripts of *Acoustics Speech and Signal Processing*, vol. ASSP-29, no. 1, p. 84, 1980.

Mark A. O'Neill is a member of a research team working on the automated production of topographical maps from satellite (SPOT) images. He is also a member of the department of photogrammetry and surveying at University College in London, U.K.